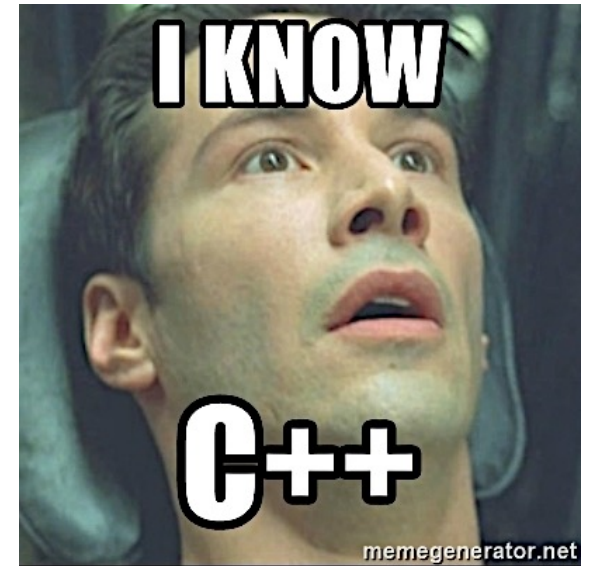
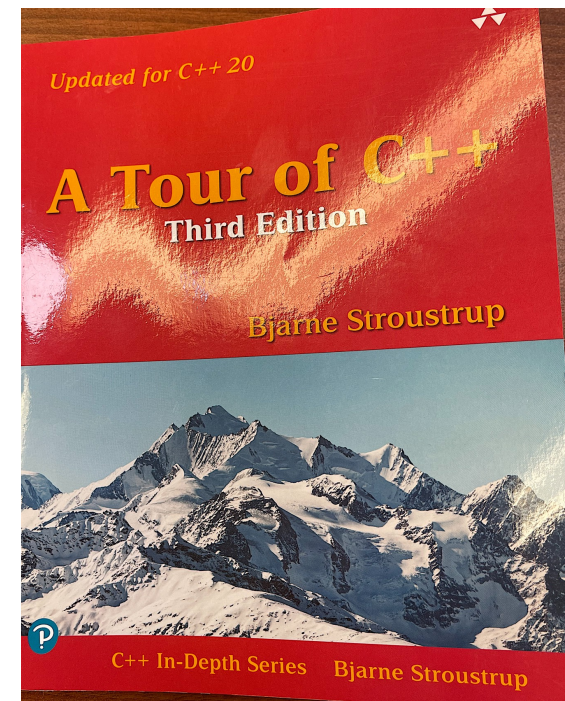


Lecture 20: C++ part V



Suggested Literature

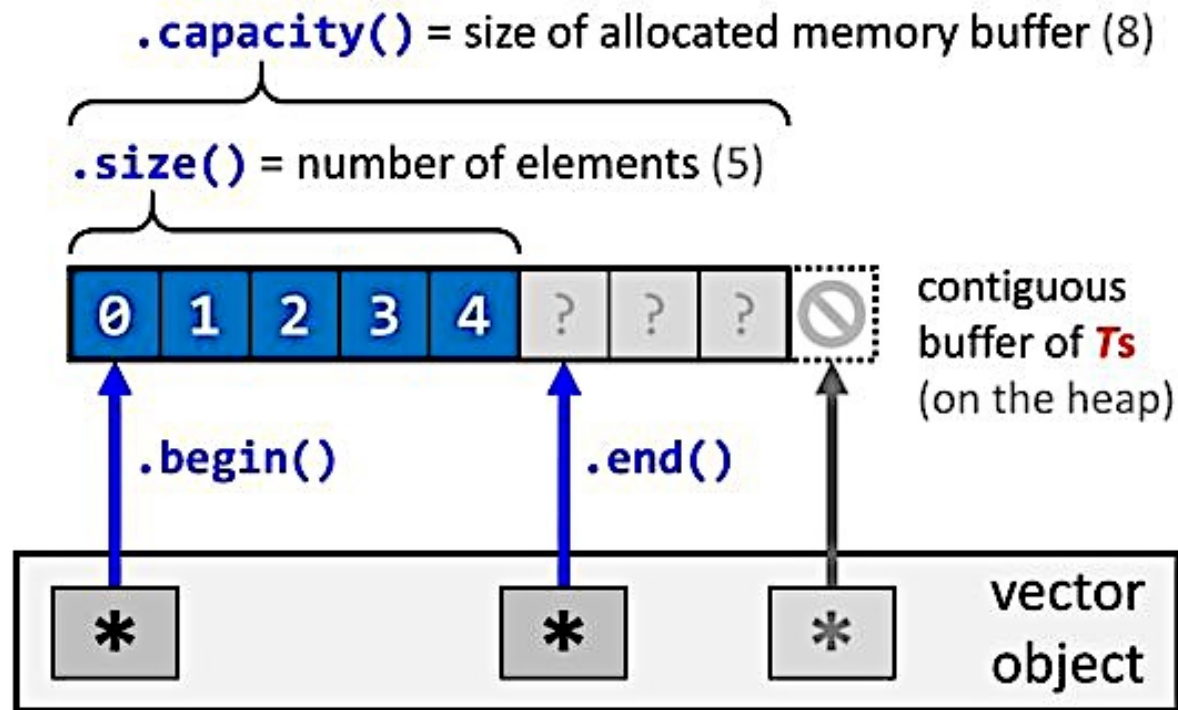


RECAP: C++ Vectors

Capacity is usually larger than size

.push_back() is cheap as long as its $\text{size} < \text{capacity}$

`std::vector<T>` Size vs. Capacity



What happens when calling **.push_back()** with **size = capacity**?

Slower: the entire vector needs to move to a new location with extra capacity

RECAP: C++ (20) ranges

```
#include<vector>
#include<print>
#include<map>
#include<ranges>
#include<string>
using std::map; using std::vector;
using std::string; using std::views::keys;
using std::ranges::to; using std::ranges::views::zip;
int main() {
    map<string,int>mymap = {{ "Vivian", 37},
                           {"Rihanna", 35}};

    // How to copy the keys of the map to a vector?
    |
    // Old C++98 style (no auto in C++98)
    std::vector<std::string> k1;
    typedef map<string,int>::iterator miter;
    for (miter it = mymap.begin(); it != mymap.end(); ++it)
        k1.push_back(it->first);

    // C++17 (structure binding)
    vector<string> k2;
    for (auto [x, _] : mymap) k2.push_back(x);

    // C++20 (views and ranges)
    vector<string> k3 = mymap | keys | to<vector>();

    // Lets print the results (C++23 style)
    for (auto [x, y, z] : zip(k1, k2, k3))
        std::println("{0}, {1}, {2}", x, y, z);
}
```

Problem: copy the keys
of a map to a vector

Old vs New C++
solutions.

This class is all about
ranges

```
whovian@phy504:~/host/Lecture19$ nano vectormap.cpp
whovian@phy504:~/host/Lecture19$ ./vmap
Rihanna, Rihanna, Rihanna
Vivian, Vivian, Vivian
```

STL Algorithm and Ranges on STL vectors

Many of them we saw in examples in the
previous three classes

Why `std::ranges` and `std::views` ?

Best answer I found came from Reddit



mredding · 2y ago

Views are cheap, non-owning accessor proxies to data. Most of the time, we want to access data, not the data container. We don't want ownership, we don't want the owner, and we don't want dependency on the owner specifics and implementation details. What are you going to do with access to vector modifying methods you don't need and didn't want? What good is are modifying methods if the vector is const? Why then are they even in scope in the first place?

https://www.reddit.com/r/cpp/comments/18ro7se/c23_views/

Cheap, non-owning accessor proxies to data

Most of the time, we want access to data, not a data container (vector, map, list...)

C++ (23) Vector (`std::ranges`)

```
#include <print>
#include <ranges>
#include <vector>

using std::views::iota; using std::views::drop;
using std::views::take; using std::print;
using std::vector; using std::ranges::to;

int main() {
    // drop(10): skips the first 10 elements
    // take(20): take only the next 20 elements, discard the rest
    auto view = iota(1, 101) | drop(10) | take(20);

    // interesting: we don't need to write the complete vector type
    // which in this case is std::vector<int>. From the iota, C++
    // can figure out that this is vector of int's.
    auto v = view | to<vector>();    ranges::to is C++23

    for (int x : v) print("{} ", x);
    print("\n");
}
```

```
whovian@whovianphy504:~/host/Lecture19$ nano vector11.cpp
whovian@whovianphy504:~/host/Lecture19$ g++ vector11.cpp -std=c++26 -o v11
whovian@whovianphy504:~/host/Lecture19$ ./v11
11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
```

As we saw
before, **C++20**
introduces
`std::ranges`
capabilities
(looks like bash)

Excellent for
filtering data

`std::view` are
ranges

C++ (23) Vector (`std::ranges`)

```
#include <print>
#include <ranges>
#include <vector>
using std::views::take;
int main() {
    std::vector Numbers{1, 2, 3, 4, 5};
    // views::take fits quite nicely with Python
    // style for loop (if you don't want to iterate
    // on the entire vector
    for (const auto num : take(Numbers, 3)) {
        std::print("{0} ", num);
    }
    std::print("\n");
}
```

```
whovian@whovianphy504:~/host/Lecture19$ nano vector12.cpp
whovian@whovianphy504:~/host/Lecture19$ g++ vector12.cpp -std=c++26 -o v12
whovian@whovianphy504:~/host/Lecture19$ ./v12
1 2 3
```

Source: <https://www.studyplan.dev/pro-cpp/views>

`std::view` is part
of C++ ranges

**Views are Non-
Owning** (they do
not own the data
they access)

C++ always tries
to avoid copies
(i.e., create
temporaries)

C++ (23) Vector (std::ranges)

```
#include <print>
#include <ranges>
#include <vector>

int main() {
    std::vector v1{4, 1, 3, 2, 5};
    auto view = std::views::take(v1, 3);

    std::println("view[0]: {0}", view[0]);

    v1[0] = 100;

    // Views are non-owning
    // Changing the elements of the
    // underlying vector also affect its views
    std::println("view[0]: {0}", view[0]);
}
```

```
whovian@whovianphy504:~/host/Lecture19$ g++ vector14.cpp -std=c++26 -o v14
whovian@whovianphy504:~/host/Lecture19$ ./v14
view[0]: 4
view[0]: 100
```

std::view is part
of C++ ranges

**Views are Non-
Owning** (they do
not own the data
they access)

Source:
<https://www.studyplan.dev/pro-cpp/views>

C++ (23) Vector (std::ranges)

My favorite aspect of
std::view is that it
preserves the original
container.

```
#include <print>
#include <ranges>
#include <vector>
#include <algorithm>
#include <fmt/ranges.h>
using std::vector; using std::views::reverse;

int main() {
    std::vector v1{1, 2, 3, 4, 5};

    // Bad: code still a bit long, plus there is
    //      an extra comma at the end
    std::ranges::for_each(
        std::views::reverse(v1),
        [](int i) { std::print("{0}, ", i); }
    );
    std::print("\n");

    // best code: println appends \n automatically
    // | operator really resemble bash!!
    std::println("{} ", v1 | reverse);

    // avoid []
    std::println("{:n}", v1 | reverse);

    // What if we want to use another symbol to
    // separate the vector elements? Here we use
    // the FMT library (and the powerful fmt::join)
    // don't mix fmt with std funcs (fmt is more powerful)
    fmt::println("{} ", fmt::join(v1 | reverse, " | "));

    // if you want the [] back
    fmt::println("[{}]", fmt::join(v1 | reverse, " | "));
}
```

```
whovian@whovianphy504:~/host/Lecture19$ g++ vector13.cpp -std=c++26 -o v13 -lfmt
whovian@whovianphy504:~/host/Lecture19$ ./v13
5, 4, 3, 2, 1,
[5, 4, 3, 2, 1]
5, 4, 3, 2, 1
5 | 4 | 3 | 2 | 1
[5 | 4 | 3 | 2 | 1]
```

Views do not create a
temporary vector

Source: <https://www.studyplan.dev/pro-cpp/views>

C++ (23) Vector (std::ranges)

```
#include <print>
#include <ranges>
#include <vector>

int main() {
    // Interesting we don't need to write
    // std::vector<int>. The <int> is deduced from the
    // integers in the initializer list.
    std::vector v1 { 1, 2, 3, 4, 5 };

    // Two equivalent ways of writing a view::filter

    auto pf = [](int i) { return i%2 == 1; };

    // (1) Without piping
    auto fview = std::views::filter(v1, pf);

    for (const auto x : fview)
        std::print("{0} ", x);
    std::print("\n");

    // (2) With piping
    auto fview2 = v1 | std::views::filter(pf);
    for (const auto x : fview2)
        std::print("{0} ", x);
    std::print("\n");
}
```

C++ piping is a clean alternative way to call views algorithms

Source: <https://www.studyplan.dev/pro-cpp/views>

```
whovian@whovianphy504:~/host/Lecture19$ g++ vector15.cpp -std=c++26 -o v15
whovian@whovianphy504:~/host/Lecture19$ ./v15
1 3 5
1 3 5
```

C++ (23) Vector (`std::ranges`)

```
#include <print>
#include <ranges>
#include <vector>

int main() {
    std::vector v1 { 1, 2, 3, 4, 5, 6, 7 };

    auto pf = [](int i) { return i%2 == 1; };

    auto fview = v1 |
        std::views::filter(pf) |
        std::views::take(3) |
        std::views::reverse;

    for (const auto x : fview)
        std::print("{0} ", x);
    std::print("\n");
}
```

C++ piping is a clean alternative way to call and concatenate the algorithms `std::ranges` and `std::views` provides

Source: <https://www.studyplan.dev/pro-cpp/views>

```
whovian@whovianphy504:~/host/Lecture19$ g++ vector16.cpp -std=c++26 -o v16
whovian@whovianphy504:~/host/Lecture19$ ./v16
5 3 1
```

std::zip for Python like zip of vectors

```
#include <ranges>
#include <vector>
#include <ranges>
#include <fmt/ranges.h>
using std::vector; using std::string; using std::views::zip;
int main() {
    using namespace std::string_literals; // sufix s C++14

    std::vector v1{1, 2, 3, 4, 5}; // weekdays

    // BE CAREFUL: vector of strings init with string
    // literals (const char*)! We explicitly typed
    // vector<string> to force the conversion between
    // const char* and C++ string
    vector<string> en{"Monday", "Tuesday",
                     "Wednesday", "Thursday", "Friday"};

    // C++14 provides an alternative way to convert
    // between string literals and C++ strings in a
    // way C++ can deduce types (using sufix s)
    vector pt{"Segunda"s, "Terça"s,
              "Quarta"s, "Quinta"s, "Sexta"s};

    // "magic code" prove both vecs are vector<string>
    static_assert(std::is_same_v<decltype(en), decltype(pt)>);

    for (auto [x, y, z] : zip(v1, en, pt)) // Python Style
        fmt::println("{0}, {1}, {2}", x, y, z);
}
```

C++ std::ranges offer
Python-style **zip**

std::views::zip
combines multiple
containers, ranges,
views into a single
range

```
whovian@whovianphy504:~/host/Lecture20$ g++ vector23.cpp -std=c++26 -o v23 -lfmt
whovian@whovianphy504:~/host/Lecture20$ ./v23
1, Monday, Segunda
2, Tuesday, Terça
3, Wednesday, Quarta
4, Thursday, Quinta
5, Friday, Sexta
```

std::transform old vs new (C++23)

```
#include <vector>
#include <algorithm>
#include <fmt/ranges.h>
```

```
int main() {
    std::vector v1{1, 2, 3, 4, 5};
    // Lambda function square (topic of next class)
    auto square = [](int x) { return x * x; };

    // Old STL Algorithm (pre-C++20). Important facts

    // (1) std::transform needs a destination vector
    // std::vector created from one non-negative argument:
    //     it creates a vector with size v1.size()
    // Unlike C arrays, the standard mandates vector elements
    // to be default initialize (in this case to zero)
    std::vector<int> v2(v1.size());

    // (2) old STL algorithm are based on the use of iterators
    // 1st arg: begin iterator of the input container
    // 2nd arg: end iterator of the input container
    // 3rd arg: begin iterator of the output container
    // 4th arg: lambda that implements the element transform
    std::transform(v1.begin(), v1.end(), v2.begin(), square);

    fmt::println("{} ", fmt::join(v2, ", "));
}
```

```
whovian@whovianphy504:~/host/Lecture19$ g++ vector17.cpp -std=c++26 -o v17 -lfmt
whovian@whovianphy504:~/host/Lecture19$ ./v17
1, 4, 9, 16, 25
```

C++ std::views

saves time via **lazy evaluation**

Not true for old

C++<20

std::transform (and algorithms)

C++<20: need a destination vector

std::transform old vs new (C++23)

```
#include <vector>
#include <algorithm>
#include <fmt/ranges.h>

int main() {
    std::vector v1{1, 2, 3, 4, 5};
    // Lambda function square (topic of next class)
    auto square = [](int x) { return x * x; };

    // Old STL Algorithm (pre-C++20). Important facts

    // (1) std::transform needs a destination vector
    std::vector<int> v2(v1.size());

    // (2) old STL algorithm are based on the use of iterators
    // 1st arg: begin iterator of the input container
    // 2nd arg: end iterator of the input container
    // 3rd arg: begin iterator of the output container
    // 4th arg: lambda that implements the element transform

    // If you want to skip the first 2 elements
    std::transform(v1.begin()+2, v1.end(), v2.begin(), square);

    fmt::println("{} ", fmt::join(v2, ", "));
}
```

C++<20 STL

algorithms are
based on iterators
("pointers")

They compute the
algorithm when
called. So, they
need a container to
hold results

```
whovian@whovianphy504:~/host/Lecture19$ g++ vector18.cpp -std=c++26 -o v18 -lfmt
whovian@whovianphy504:~/host/Lecture19$ ./v18
9, 16, 25, 0, 0
```

std::transform old vs new (C++23)

```
#include <vector>
#include <algorithm>
#include <fmt/ranges.h>

int main() {
    std::vector v1{1, 2, 3, 4, 5};
    // Lambda function square (topic of next class)
    auto square = [](int x) { return x * x; };

    // Old STL Algorithm (pre-C++20). Important facts

    // (2) old STL algorithm are based on the use of iterators
    // 1st arg: begin iterator of the input container
    // 2nd arg: end iterator of the input container
    // 3rd arg: begin iterator of the output container
    // 4th arg: lambda that implements the element transform

    // If you want to modify v1, you can send v1.begin()
    // in the third arg. std::transform do the transformation
    // in sequence, so it is ok
    std::transform(v1.begin(), v1.end(), v1.begin(), square);

    fmt::println("{} ", fmt::join(v1, ", "));
}
```

C++<20 STL

algorithms: based
on iterators.

Results need a
container.

Can the output
vector be **v1** itself?

Yes. But that is
destructive.

```
whovian@whovianphy504:~/host/Lecture20$ g++ vector19.cpp -std=c++26 -o v19 -lfmt
whovian@whovianphy504:~/host/Lecture20$ ./v19
1, 4, 9, 16, 25
```

std::transform old vs new (C++23)

```
#include <vector>
#include <algorithm>
#include <fmt/ranges.h>

int main() {
    std::vector v1{1, 2, 3, 4, 5};
    // Lambda function square (topic of next class)
    auto square = [](int x) { return x * x; };

    // (2) old STL algorithm are based on the use of iterators
    // 1st arg: begin iterator of the input container
    // 2nd arg: end iterator of the input container
    // 3rd arg: begin iterator of the output container
    // 4th arg: lambda that implements the element transform

    // If you want to modify v1, you can send v1.begin()
    // in the third arg. BE CAREFUL.
    // THE v1 output will be quite messy because of the +2
    std::transform(v1.begin()+2, v1.end(), v1.begin(), square);

    fmt::println("{} ", fmt::join(v1, ", "));
}
```

C++<20 STL

algorithms: can the
output vector be
v1 itself? Yes. But
that is destructive.

Be careful with **v1**
being the output
itself!

```
whovian@whovianphy504:~/host/Lecture20$ g++ vector20.cpp -std=c++26 -o v20 -lfmt
whovian@whovianphy504:~/host/Lecture20$ ./v20
9, 16, 25, 4, 5
```

std::transform old vs new (C++23)

```
#include <vector>
#include <algorithm>
#include <fmt/ranges.h>

int main() {
    std::vector v1{1, 2, 3, 4, 5};
    // Lambda function square (topic of next class)
    auto square = [](int x) { return x * x; };

    // (2) old STL algorithm are based on the use of iterators
    // 1st arg: begin iterator of the input container
    // 2nd arg: end iterator of the input container
    // 3rd arg: begin iterator of the output container
    // 4th arg: lambda that implements the element transform

    // If you want to modify v1, you can send v1.begin()
    // in the third arg. BE CAREFUL. you must match
    // the iterator positions carefully
    std::transform(v1.begin()+2, v1.end(), v1.begin()+2, square);

    fmt::println("{} ", fmt::join(v1, ", "));
}
```

C++<20 STL

algorithms: be
careful with **v1**
being the output
itself!

You must match
the iterators
carefully

```
whovian@whovianphy504:~/host/Lecture20$ g++ vector21.cpp -std=c++26 -o v21 -lfmt
whovian@whovianphy504:~/host/Lecture20$ ./v21
1, 2, 9, 16, 25
```

std::transform old vs new (C++23)

```
#include <algorithm>
#include <fmt/core.h>
#include <ranges>
#include <vector>
#include <fmt/ranges.h>
using std::views::iota; using std::ranges::to;
using std::sqrt; using std::sin; using std::back_inserter;
using std::transform; using std::vector;
int main () {
    // initialize a vector with std::views. Remember that
    // views have lazy-evaluation. Don't forget. ALWAYS
    // need the std::ranges::to to copy view to a container
    vector<int> v1 = iota(1, 5) | to<vector>();

    auto pf = [](int n) { return 2*n; };

    // OLD C++98.
    // What if you have not allocated vout? Need inserter
    // back_inserter is a type of inserter - see last lecture)
    vector<int> v2;
    transform(v1.begin(), v1.end(), back_inserter(v2), pf);

    // Here we allocate vout2 - so no inserter needed
    vector<int> v3;
    v3.resize(5);
    transform(v1.begin(), v1.end(), v3.begin(), pf);

    fmt::println("{} ", fmt::join(std::views::zip(v2,v3), ", "));
}
```

In old C++98,

std::transform (which
uses iterators).

What if we don't pre-
allocate memory on the
output vector?

STL inserters come to
the rescue

```
whovian@phy504:~/host/Lecture20$ nano vector30.cpp
whovian@phy504:~/host/Lecture20$ g++ vector30.cpp -o v30 -std=c++26 -lfmt
whovian@phy504:~/host/Lecture20$ ./v30
(2, 2), (4, 4), (6, 6), (8, 8)
```

std::view vectors and lazy-evaluation

```
#include <vector>
#include <algorithm>
#include <ranges>
#include <fmt/ranges.h>

int main() {
    std::vector v1{1, 2, 3, 4, 5};
    // Lambda function square (topic of next class)
    auto square = [](int x) { return x * x; };

    // the computation DOES NOT HAPPEN here
    // this is similar to Python generators
    auto squares = v1 | std::views::transform(square);

    // computation happens here, when squares output
    // is needed (again similar to Python generators)
    // Important fact: computation happens ON THE FLY
    // therefore, no additional container is needed
    fmt::println("{} ", fmt::join(squares, ", "));
}
```

C++ std::views

saves time via **lazy evaluation**

No need for an output container.

Key: understand **WHEN** the transformation is being computed

```
whovian@whovianphy504:~/host/Lecture20$ g++ vector22.cpp -std=c++26 -o v22 -lfmt
whovian@whovianphy504:~/host/Lecture20$ ./v22
1, 4, 9, 16, 25
```

Combining new and old styles

```
#include <vector>
#include <algorithm>
#include <ranges>
#include <numeric>
#include <fmt/ranges.h>
using std::views::iota; using std::views::reverse;
using std::ranges::to; using std::vector;
using std::views::common; using std::sort;
int main() {
    vector v1 = iota(1,10) | to<vector>();

    auto square = [](int x) { return x * x; };
    auto squares = v1 | std::views::transform(square);

    fmt::println("{} ", fmt::join(squares, ", ")); // print
                                                    // squares

    // to send squares to C++98 STL (read only) algo: common
    auto v2 = squares | common;
    int sum = std::accumulate(v2.begin(), v2.end(), 0);
    fmt::println("{} ", sum);
}
```

- **C++98**: based on iterators (pointers)
- **C++20**: based on ranges

Can we mix the old-style algorithm with a new one? Yes, if it is read-only usage
(**views::common**)

```
whovian@phy504:~/host/Lecture20$ nano vector32.cpp
whovian@phy504:~/host/Lecture20$ g++ vector32.cpp -o v32 -std=c++26 -lfmt
whovian@phy504:~/host/Lecture20$ ./v32
1, 4, 9, 16, 25, 36, 49, 64, 81
285
```

std::transform old vs new (C++23)

```
#include <vector>
#include <algorithm>
#include <fmt/ranges.h>
```

```
int main() {
    std::vector v1{1, 2, 3, 4, 5};
    std::vector v2{5, 4, 3, 2, 1};
    // Lambda function acting as a binary operator
    auto pf = [](int x, int y) { return x + y; };

    // (2) old STL algorithm are based on the use of iterators
    // 1st arg: begin iterator of the input (1) container
    // 2nd arg: end iterator of the input (1) container
    // 3rd arg: begin iterator of the input (2) container
    // 4th arg: begin iterator of the output container
    // 6th arg: lambda that implements the element transform

    // old STL transform needs a destination vector fully allocated
    std::vector<int> out(v1.size()); // here we need to fully
                                     // specify this is a vector
                                     // of ints - there is nowhere
                                     // C++ can deduce this info

    // OLD algorithm style
    std::transform(v1.begin(), v1.end(),
                  v2.begin(), out.begin(), pf);

    fmt::println("{} ", fmt::join(out, ", "));
}
```

std::transform is
also compatible
with binary
operators (if we
want, for instance,
to sum two vectors)

```
whovian@whovianphy504:~/host/Lecture20$ g++ vector24.cpp -std=c++26 -o v24 -lfmt
whovian@whovianphy504:~/host/Lecture20$ ./v24
6, 6, 6, 6, 6
```

std::transform old vs new (C++23)

```
#include <vector>
#include <ranges>
#include <fmt/ranges.h>
using std::views::zip_transform;
int main() {
    std::vector v1{1, 2, 3, 4, 5};
    std::vector v2{5, 4, 3, 2, 1};
    // Lambda function acting as a binary operator
    auto pf = [](int x, int y) { return x + y; };

    // New C++23 style
    auto view = zip_transform(pf, v1, v2);
    fmt::println("{} ", fmt::join(view, ", "));

    // C++23 is more powerful (allows >2 args)
    auto pf2 = [](int x, int y, int z) { return x + y + z; };
    std::vector v3{1, 1, 1, 1, 1};
    auto view2 = zip_transform(pf2, v1, v2, v3);
    fmt::println("{} ", fmt::join(view2, ", "));
}
```

std::transform is

also compatible
with binary
operators

New C++23
syntax allows >2
arguments

```
whovian@whovianphy504:~/host/Lecture20$ g++ vector25.cpp -std=c++26 -o v25 -lfmt
whovian@whovianphy504:~/host/Lecture20$ ./v25
6, 6, 6, 6, 6
7, 7, 7, 7, 7
```

std::transform old vs new (C++23)

```
#include <vector>
#include <ranges>
#include <fmt/ranges.h>
using std::views::zip_transform;
int main() {
    std::vector v1{1, 2, 3, 4, 5};
    std::vector v2{5, 4, 3, 2, 1};
    std::vector v3{1, 1, 1, 1, 1};

    // Variadic Lambda
    // PS: auto on arg here is mandatory
    auto pf = [](auto... x) { return (x + ...); };

    auto view = zip_transform(pf, v1, v2);
    fmt::println("{} ", fmt::join(view, ", "));

    auto view2 = zip_transform(pf, v1, v2, v3);
    fmt::println("{} ", fmt::join(view2, ", "));
}
```

We are going to see
next class (on
lambdas) that
C++>17 allows
variadics (variable
number of
arguments)

```
whovian@whovianphy504:~/host/Lecture20$ g++ vector26.cpp -std=c++26 -o v26 -lfmt
whovian@whovianphy504:~/host/Lecture20$ ./v26
6, 6, 6, 6, 6
7, 7, 7, 7, 7
```

std::accumulate old vs new (C++23)

```
#include <vector>
#include <algorithm> // fold_left
#include <numeric>    // old accumulate
#include <print>
int main() {
    std::vector v1{1, 2, 3, 4, 5, 6, 7};

    // OLD STL WAY: in both old and new algorithm
    // the 0 instead of 0.0 is quite important.
    // Why? it will dictate the return type.
    // That is the where auto gets the type information
    auto res1 = std::accumulate(v1.begin(), v1.end(), 0);

    // New STL way
    // some trivial binary operators are already in the STL
    auto res2 = std::ranges::fold_left(v1, 0, std::plus{});
    std::println("{0} and {1}", res1, res2);
}
```

One
particularly
powerful STL
algorithm is
std::accumulate

```
whovian@whovianphy504:~/host/Lecture20$ g++ vector27.cpp -std=c++26 -o v27 -lfmt
whovian@whovianphy504:~/host/Lecture20$ ./v27
28 and 28
```

std::accumulate old vs new (C++23)

```
#include <vector>
#include <algorithm> // fold_left
#include <numeric>    // old accumulate
#include <cmath> // large number of special funtions
#include <print>
int main() {
    std::vector v1{1, 2, 3, 4, 5, 6, 7};

    // custom binary op
    auto pf = [](double x, double y) {
        auto a = std::legendre(2, x); // legendre polynomial
        auto b = std::laguerre(3, y); // laguerre polynomial
        return a+b;
    };

    auto res1 = std::accumulate(v1.begin(), v1.end(), 0., pf);

    auto res2 = std::ranges::fold_left(v1, 0., pf);

    std::println("{0} and {1}", res1, res2);
}
```

std::accumulate
ok with custom
binary operator.

Powerful when
combining with
C++17 lib of
special
functions

```
whovian@whovianphy504:~/host/Lecture20$ g++ vector28.cpp -std=c++26 -o v28 -lfmt
whovian@whovianphy504:~/host/Lecture20$ ./v28
11895818481.718742 and 11895818481.718742
```

std::accumulate old vs new (C++23)

```
#include <random>
#include <fmt/core.h>
#include <ranges>
#include <cmath> // large number of special functions
#include <range/v3/all.hpp>
#include <fmt/ranges.h>
int main () {
    std::vector v1 = {1, 2, 3, 4, 5, 6, 7};
    std::vector v2 = {7, 6, 5, 4, 3, 2, 1};

    auto pf = [](int x, int y) { // custom binary op
        return x*x + y*y;
    };

    // Remember: views are lazy-eval.
    auto view = std::views::zip_transform(pf, v1, v2);

    // So what do we do with transform view? E.g., Print
    fmt::println("{} ", fmt::join(view, ", "));

    // ranges-v3: we can accumulate all transform view elements
    auto res = ranges::accumulate(view, 0); // from ranges-v3

    fmt::println("Acc = {}", res);
}
```

std::fold_left is not
as powerful as it
could be

Extensions to STL
can be found in
[ranges-v3](#)

```
whovian@whovianphy504:~/host/Lecture20$ g++ vector29.cpp -std=c++26 -o v29 -lfmt -I /vcpkg/installed/x64-linux/include/
-Wno-deprecated-literal-operator
whovian@whovianphy504:~/host/Lecture20$ ./v29
50, 40, 34, 32, 34, 40, 50
Acc = 280
```

Homework I

Given two STL vectors **v1** and **v2** of the same size.

- (1) Create vectors v1 and v2 with a size of 1,000,000. Fill them with random numbers computed using the STL library random number generators from a normal distribution of mean 3.0 and standard deviation 1.0. Use C-style loop. Don't allow zero values.
- (2) Compute their sum using **std::transform** algorithm (use **std::plus** and not lambda to teach transform how to sum. Then try again using lambda). Try with serial and parallel execution.
- (3) Compute v1/v2 using **std::transform** algorithm (use **std::divides** and not lambda to teach transform how to divide. Then try again using lambda). Try with serial and parallel execution.
- (4) Repeat (2) and (3) using C style for loops. And python loops. Copy by value and l-value reference
- (5) Repeat (2) & (3) but use **std::ranges::for_each** and vector **.push_back()**
- (6) Repeat (2-3-4-5), but use **std::chrono** to time all the executions.

Homework II

Given two STL `std::vectors`, `v1` and `v2`, of the same size.

- (1) Create vectors `v1` and `v2` with a size of 1,000,000. Fill them with random numbers computed using the STL library random number generators from a Bernoulli distribution with $p = 0.25$. Use `std::ranges::generate` to fill `v1` and `v2`
- (2) Calculate their sum and multiplication. Use `std::multiplies`
- (3) Using both `std::ranges::for_each` + lambda function and stream (`cout <<`), print the first 100 elements of (1) and (2). Do not let the comma go past the last element. (a mistake I made in a lot of slides)
- (4) Repeat (1) but adopt `std::ranges::for_each`

Homework III

Given a STL `std::vector`, **v1** and `std::list` **v2**, of the same size.

- (1) Create `std::vector` v1 and `std::list` v2 with a size of 1,000,000. Fill them with random numbers computed using the STL library random number generators from a Poisson distribution with $\lambda = 5$. However, use `push_back` to fill them without pre-allocating capacity. Use `std::chrono` to determine which one is faster.
- (2) Now, pre-allocate the vector capacity. What is the new time?
- (3) Using `std::min_element` compute the minimum element of the arrays
- (4) Using `std::inner_product` compute the norm of the `std::vector`
- (5) Using `std::accumulate` and **lambdas**, compute the norm of the `std::vector`. Do not create a temporary third vector by doing that
- (6) Now use `std::transform_reduce`, try both sequential and parallel execution

Homework IV

In the example where we copy all the keys of an `std::map` to a `std::vector`, we used the `std::views::key`

- (1) Similar to `std::views::key`, the STL provides `std::views::values`. Use that to construct a full example where we copy all the values held by keys in a `std::map<string,int>` to a `std::vector<int>` (do it in one line!)
- (2) Repeat (1) using C++17 structure binding. Do not allocate wasteful variable for the map's key
- (3) Repeat (1) using C++98 `std::map<string,int>::iterators`. Do (3) in 2 ways.
 - (1) Using `std::vector::push_back()` (extra points if you pre-allocate the vector capacity)
 - (2) Using operator `[]` and pre-allocating vector size;
- (4) Similar to `std::views::key`, the STL provides `std::views::elements`. Use that view to construct a complete example where we copy all the {key, value} pair elements of a map into an `std::vector`
- (5) Repeat (4) using C++17 structure binding. Extra point if you also print the pairs using `std::println`
- (6) Repeat (4) using C++98 `std::map<string,int>::iterators`